THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Guarding the Boundary: Information Flow Tracking in the Presence of Libraries

ALEXANDER SJÖSTEN

CHALMERS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Guarding the Boundary: Information Flow Tracking in the Presence of Libraries
ALEXANDER SJÖSTEN

Technical report 175L
ISSN 1652-876X
Department of Computer Science and Engineering
Information Security Division

CHALMERS UNIVERSITY OF TECHNOLOGY
SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 10 00

Printed at Chalmers
Gothenburg, Sweden 2018

Guarding the Boundary: Information Flow Tracking in the Presence of Libraries

Alexander Sjösten
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

In modern software development, the use of libraries is prevalent. Libraries pose a big security challenge. How can we ensure that sensitive data is not being leaked through libraries? This is the first question of the thesis. We propose the use of information-flow control, by developing a principled approach for allowing information-flow tracking in libraries, even if they are written in a language not supporting information-flow control. With this approach, we allow for library functions to have *unlabel* and *relabel* models, explaining how values are unlabeled and relabeled when being marshaled between the labeled program and the library. These models are used in combination with *lazy marshaling* to handle structured data such as lists and records, higher-order functions and references.

Modern browsers allow for browser modifications through *browser extensions*, which have special privileges and can, e.g., modify the DOM. As extensions can be intrusive, it is in a webpage's interest to know which extensions are installed in a browser. The second question of the thesis is if it is possible for a webpage to know which extensions are installed in the browser? We conduct a large-scale study to determine how many extensions that are detectable from a webpage based on the extension's resources, showing over 50% of the top 1000 Chrome extensions can be detected, as well as how many of the Alexa top 100,000 webpages employ the technique of the paper.

Keywords: information-flow control, language-based security, side-effectful libraries, web security, browser extensions, large-scale study

Acknowledgements

First and foremost, I want to thank my supervisor Andrei Sabelfeld for all the support, great advice about coffee places, lunch runs and tips about the academic life. Without his support, this work would not have been possible.

I also want to thank my co-supervisor Daniel Hedin for great collaborations, advice, gaming nights, beers, coffees and for leading me down the path to frustration. Praise the Sun, Daniel!

Thank you Steven, who was the co-author on one of the papers for all the advice you have given me about organising the research.

A big thank to my office mates, both old and new, for all the discussions and for making the office a truly awesome working environment: Pablo, Daniel S, Per, Jeff and Iulia.

To all my other colleagues at Chalmers, my deepest and sincerest thank you for all the coffees, beers, lunches and dinners. You all help make Chalmers such a nice place to work at, and I count you all as not just colleagues, but also friends.

To my former study mates from the bachelor's and master's education whom I still keep contact with, thank you for sticking around and sharing coffees, lunches, gaming nights, hockey nights, role playing sessions and whatnot. You help disconnecting the brain from work and I simply have a great time hanging out with you.

To my family, thank you so much for all encouragement and support you have given throughout the years. Although we have had some disagreements (it is what a son and brother is for, right?), you have always been there when I needed it the most, just a phone call away.

Last, but definitely not least: a big thanks goes to Pauline. I don't understand how you can stand to live with me, but you give me so much love and support. You are always there to pick me up when I fall down, supporting me every step I take. You are truly awesome!

Contents

Guarding the Boundary: Information Flow Tracking in the Presence of Libraries

In society today, most business sectors are completely reliant on information technology, and our day-to-day lives are moving online at an astonishing pace. For example, we use computers to talk to friends, read newspapers, watch movies, schedule events, make bank transfers, and buy merchandise. With more and more of our private information going online, the need to protect this information increases.

Unfortunately, it is hard to ensure that private information is not leaked to unintended recipients – even for domain experts. Adding a simple feature, for example tracking how users use a web application, can lead to private information being leaked [28]. In recent years, there have been many reported breaches of security in big companies, where sensitive information, such as credit card information, passwords and emails, has been stolen [13, 3, 27, 22, 26, 21, 24, 8], leading to financial losses and even loss of life [7].

A difficult problem to handle in modern applications is the use of *libraries*. With growing code bases, and potentially different languages getting access to sensitive information, securing the boundaries between a program and its libraries is a big challenge. This challenge is the focus of the first two papers of this thesis: tracking how information is flowing in a library used by a trusted program.

With services coming in the shapes of web applications, we need *web browsers* to access these services. But each individual has their own preferences on how the browsing should be. The users are therefore given the opportunity to extend the browser with functionality via *browser extensions*. There are cases where webpages try to determine what extensions are running, as the existence of an extension can lead to, for example, financial losses due to ad blockers. Another reason can be because the webpage want a clean environment due to handling of sensitive information, or because the webpage is malicious and tries to fingerprint a user. The main goal of the third paper of this thesis is to determine how many extensions for Chrome and Firefox are susceptible to webpages trying to determine their existence.

The first two papers in this thesis are theoretical, whereas the third paper is practical. The future goal is to enable end-to-end security for web applications in the presence of libraries. Since e.g. extensions can manipulate the DOM using JavaScript, "libraries" in this setting corresponds to the DOM API in the browser.

The rest of this chapter is laid out as follows. Section 1 gives an introduction to *information-flow control*, which is the main security mechanism used in this thesis. Section 2 gives a brief introduction to the information security challenges of libraries. Section 3 gives an introduction to browser extensions before Section 4 lists the main contributions of the papers, along with the

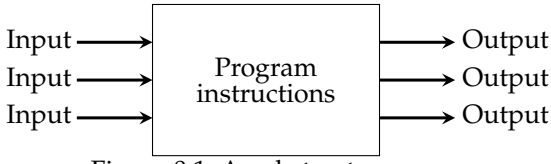

Figure 0.1: An abstract program

contributions made by the author.

1 Information-Flow Control

In software development, the most common ways of checking if an application is correct is through testing and code reviews. Although testing and code reviews can find some security vulnerabilities, many are missed; see for instance Heartbleed [33] and Shellshock [32].

Language-based security is a means to express security policies and enforcement mechanisms using programming language techniques [30]. Frequently, the goal of language-based security is to be provably secure. In this thesis, we will focus on the area of language-based security called *information-flow control (IFC)*. As is common, we work in a batch model of programs, meaning that we see programs as black boxes, treating them like functions from input to output (see Figure 0.1). We call the *inputs* to the program *sources*, and the *outputs* of the program *sinks*. For all useful programs, the outputs are dependent on the inputs. The dependencies from sources to sinks are defined by the program source code, which is written in some programming language.

In a *multi-level system* [9], information is classified into different *levels*, based on a *lattice*. Typical example levels are *unclassified* $\sqsubseteq$ *classified* $\sqsubseteq$ *secret* $\sqsubseteq$ *top secret*, where $\sqsubseteq$ is a relation defining how information is allowed to flow. In the example above, *unclassified* data can flow anywhere, and *classified* data can flow anywhere but to *unclassified*. IFC aims to enforce that the information flow respects the $\sqsubseteq$-relation. Without loss of generality, we use a two-level lattice, $L \sqsubseteq H$, where L is *public* (low) data and H is *secret* (high) data. In this simplified setting, the security property we want to enforce is that sources marked as H does not go to sinks marked as L, which is known as noninterference [14].

1.1 Noninterference

Intuitively, noninterference is achieved if all runs of a program, where the only difference is the high inputs, do not differ in the low output. This means the crossed out dashed red line in Figure 0.2 is not allowed.

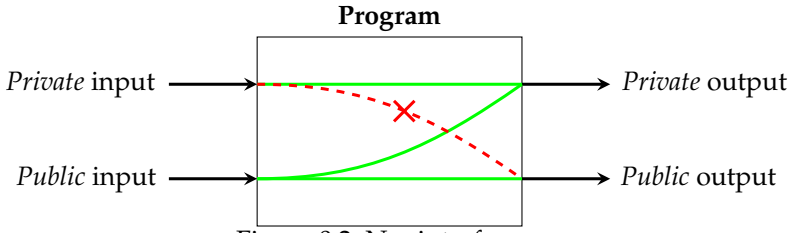

Figure 0.2: Noninterference

In this thesis, we will only consider *termination-insensitive noninterference (TINI)* [31]. The implication of enforcement through TINI is that information leakage through *termination channels* is not considered. Assuming `hi` is an integer labeled H and `print` outputs on a public channel, the following program is deemed secure by state-of-the-art IFC tools that do not take termination into account.

```
1  for i in range(0, Number.MAX_VALUE) {
2      print(i);
3      if (i == hi) then loop_forever
4  }
```

As the for-loop is not dependent on the secret variable, the output on the public channel is allowed. However, once `i == hi`, the program will be stuck in an infinite loop, ensuring the final printed value on the public channel to be the secret value, indicating there exists an implicit flow through a termination channel [4]. TINI gives no guarantees about non-terminating runs, since it is formulated in terms of terminating runs. Hence, TINI would be unable to classify the program above as insecure.

1.2 Explicit and implicit flows

In order to achieve noninterference, we must track how information flows within the program. There are two different kinds of flows: *explicit* and *implicit* flows. An example of an explicit flow is when secret data is written directly to a public sink or variable. Imagine two variables, `lo`, which is labeled L, and `hi`, which is labeled H. An explicit flow would be `lo := hi`, as the secret data in `hi` is written directly to the public variable `lo`. In contrast to explicit flows, an implicit flow has no data being written directly from a secret to a public sink or variable. Instead, the control flow of the application is used to learn something about the secret data. As an example, consider the following program that leaks if the variable `hi` is even or odd through the control flow.

```
1 lo := false;
2 if (hi `mod` 2 == 0) then lo := true;
```

Note that although no information about the secret variable `hi` is explicitly written to the public variable `lo`, we can still learn information about it.

1.3 Enforcing Information-Flow Control

Within IFC, there are two different main approaches of enforcement [29]. On one hand, there is *static* enforcement, which is based on static analysis of a program before it is executed. Volpano et al. [34] presented a *type system* with the property that all well-typed programs in this system are sound with respect to noninterference.

On the other hand, *dynamic* enforcement is executed at run-time, using a modified semantics of the language to allow for security checking. Having full access to the run-time environment and the run-time values, dynamic enforcement often leads to a more permissive enforcement compared to the static counterparts for dynamic languages such as JavaScript.

Additionally, there exists combinations of the static and dynamic enforcements known as *hybrid* enforcement. With hybrid enforcement, a static enforcement mechanism can insert annotations during the compilation phase, which can be checked at runtime [11, 12]. Similarly, a dynamic enforcement mechanism can perform static analyses on parts of the program during execution [19].

1.4 Dynamic Information-Flow Control

This thesis is based on purely dynamic IFC. The main reason we chose dynamic IFC is because dynamic languages, such as JavaScript, are widespread and popular, especially on the web.

In dynamic IFC, all runtime values are augmented with a representation of security labels. These labels are copied and joined to reflect the different computations of the program. During execution, a *program counter (pc)* is used to keep track of which level the current execution occurs in (known as the *security context*). If secret data is used when computing the condition in an if-statement, the pc is updated to reflect this, and the body of the if-statement is executed under *secret control*. While under secret control, no public side-effects should be allowed to take place. It is crucial that the handling of side-effects under secret control is done in a safe manner. Otherwise there is a risk of implicit flows into the labels. Consider the following example from [6], where `lo` and `tm` are labeled L and `hi` is labeled H at the start of the execution. What would be the implications of allowing labels to change

freely, i.e. upgrading the security labels on the assigned variables on lines 3 and 4 if the assignment occurs?

```
1 lo := true;
2 tm := true;
3 if (hi == true) then tm := false;
4 if (tm == true) then lo := false;
```

If `hi` is `true`, then `tm` will be `false` based on the assignment on line 3, and its security label upgraded to H, due to the pc being H. Since `tm` is `false`, the condition in the if-statement on line 4 will be `false`, making no assignment to `lo`, which means `lo` will continue to be `true` and labeled L. But if `hi` was `false`, then no assignment would be made to `tm` on line 3, making `tm` remain `true` and labeled L. This would make the assignment to `lo` on line 4 occur in a low context due to the pc being L , making `lo false` and labeled L. The end result in both situations is the value of `hi` being the same as `lo`, while `lo` retains the label L. In other words, `hi` was leaked into `lo`!

The most direct way of preventing the problem in the previous example and avoiding the implicit flows into the labels is to base the enforcement on *no sensitive-upgrades (NSU)*, which disallows upgrading low variables when branching on secret data [5, 35]. With NSU, the assignment on line 3 would not be allowed, as there is a low upgrade under high control, causing a termination of the program before the information leakage occurs.

A problem with NSU is that it can sometimes be too restrictive and mark valid programs as invalid. One could argue the program

```
1 lo := true;
2 if (hi == false) then lo := false;
```

is secure if the low variable `lo` is never written to a public sink, hence not visible to an attacker. The program

```
1 lo := false;
2 if (hi == false) then lo := false;
```

can also be deemed secure, since an attacker will not learn anything about `hi` since the value of `lo` is never changed. This is known as value sensitivity [10] and is not in scope of this thesis. The enforcement mechanisms of this thesis would consider those programs insecure if `hi` is indeed `false`.

2 Libraries

One focus of this thesis is how to handle libraries with respect to information-flow tracking. A major challenge is when the library is not written in the same language as the labeled program. This usually happens in one of two

cases: 1) the library is part of the standard execution environment, and 2) the library is brought into the language using a *foreign function interface (FFI)*. An FFI can therefore be used to extend a programming language with features, such as network communication, not natively in the programming language. When that happens, the values going between the labeled program and the library must be translated, a process known as *marshaling*. This poses a big challenge, since the security labels from the labeled program values must be removed when values go into the library, and put back when values are going from the library to the labeled program. Unlike standard marshaling, which is lossless, this means that we lose data when marshaling between a labeled program and an unlabeled library; we will not be able to compute the labels of the return values without knowing the labels of the arguments. To solve this, we need to keep a *state* in the marshaling process, where we store the labels that are removed. The labels are stored with respect to a *function model* for the library function, which defines how labels are removed (through an *unlabel model*) and how they are re-attached (through a *relabel model*). This interaction is what Papers I and II address.

2.1 Lazy marshaling

When marshaling structured values, such as lists, the question of how to do it effectively arises. It can be done strictly, where the full value is marshaled directly. This is expensive for large structured data, as we might marshal more than is needed for the computation. Another way is to do it lazily, which allows for marshaling on a need basis. With *lazy marshaling*, only the traversed elements of the structured data affect the computed return label. Imagine having a list of ten elements, but only the first two are needed for the computation. Strict marshaling would marshal the full list, but lazy marshaling would marshal exactly the two needed elements.

This notion of lazy marshaling extends naturally to all types of structured data, including records and objects as well.

3 Browser extensions

In order to increase web browser functionality, users install browser extensions. Examples of browser extensions are ad blockers to block advertisement on webpages, anti-tracking extensions to avoid tracking from tracking software, and password managers to make it easier to have unique passwords for all services. But it comes at a cost, as extensions are given permissions greater than those of a webpage. For instance, an extension can inject arbitrary code [16], with some malicious extensions actually injecting tracking software to track their users on every webpage they visit [18]. Even worse, if

an extension has a vulnerability, it might allow webpages to execute arbitrary code with elevated privileges [2, 1]. It is in a webpage's interest to know which extensions a user has installed, as it can lead to, for instance, financial losses due to less advertisements being showed, or to prevent arbitrary code being injected when paying bills over an internet bank.

Webpages can detect extensions using *behavioral analysis*, where one tries to detect extensions by looking for the effects of the extension. An example of this would be to check for the presence or absence of elements on the webpage. It is, however, difficult to use behavioral analysis to identify a specific extension – there are, for instance, several different ad blockers that have the same behavior. Behavioral analysis is also costly, as it requires time and effort to analyse and keep up-to-date with extension updates. Is there an easier way to determine which extensions a user has installed? This is the question tackled in Paper III.

4 Contributions

This thesis presents three papers, where Paper I and Paper III are published in peer-reviewed conferences. Paper II is currently under submission. All three papers presented are extended versions.

4.1 A Principled Approach to Tracking Information Flow in the Presence of Libraries

In this paper, we explore and develop an approach for tracking the information flow in a program that uses libraries. The program is assumed to be written in an information-flow aware language, whereas the library is not. The development is made gradually, starting with a small core language which is then extended with lists and higher-order functions. The general idea is based on *unlabel* and *relabel* models, which defines how labels are removed when marshaling to the library, and how labels are re-attached when marshaling back to the labeled world. An important part in the paper is the *lazy marshaling*, which increases the precision in the tracking as only the used parts of for example a list will affect the resulting label when marshaling back to the labeled world. The system is proven sound with respect to noninterference.

Statement of contribution This paper was co-authored with Daniel Hedin, Frank Piessens and Andrei Sabelfeld. Alexander's contributions was to define the syntax and semantics together with Daniel, implement prototypes

for testing the ideas, and prove soundness of the different systems. All authors contributed to the writing of the paper equally.

Appeared in *Principles of Security and Trust (POST), Uppsala, Sweden, April 2017*

4.2 Information Flow Tracking for Side-effectful Libraries

The second paper of the thesis is a continuation of the first paper, where the major contribution is the addition of references and with that, side-effects. The core system was overhauled, introducing a *model heap* instead of passing the model state as an (implicit) parameter. With the use of a persistent heap structure, the list and higher-order functions from Paper I had to be modified, along with an extension of records, references and side-effects. The lazy marshaling remained for lists, as well as for the newly implemented records, but the model language was extended to also contain *side-effect constraints*, where the side-effect constraints are models for how the side-effects can manipulate data. The theoretical work is formalised in Coq [20], showing the system is sound with respect to noninterference.

This work, along with Paper I provides a core for how to track information flow in stateful libraries with structured data and higher-order functions.

Statement of contribution This paper was co-authored with Daniel Hedin and Andrei Sabelfeld. Alexander's contributions was to define the syntax and semantics, conduct the case study on a file system library, creating the examples and implementing the prototype. He also wrote an initial version of the paper, which served as the base when turning it to a coherent paper. All authors contributed equally in the latter process.

4.3 Discovering Browser Extensions via Web Accessible Resources

Webpages can perform browser fingerprinting, by detecting specific configurations of the hardware, see for instance Panopticlick [25]. But can extensions be detected by webpages, without the need of analysing their behavior? The third paper explores what knowledge can be gained from a webpage about a user's installed browser extensions. It takes advantage of *web accessible resources (WARs)*, which are public resources for an extension [17]. These WARs can be fetched from any webpage, indicating that all extensions that have at least one listed WAR can easily be detected.

This work includes downloading all free extensions for Chrome [15] and Mozilla [23], as well as crawling the Alexa top 100,000 pages and analyse the requests made, to determine if this technique is widely used. It also includes potential measures one can implement in order to avoid this kind of detection.

Statement of contribution This paper was co-authored with Steven Van Acker and Andrei Sabelfeld. Alexander's contributions was the extensions experiment (all but the Alexa part), as well as defining the measures and develop the prototype for detecting extensions. All authors contributed equally to writing the paper.

Appeared in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY), Scottsdale, AZ, USA, March 2017*

5 Differences between Paper I and Paper II

Although the first two papers of this thesis handle the same topic, there are differences between them. At a high level, Paper II is a superset of Paper I, where the main difference is the addition of more language features, such as records and references. However, when looking at the semantic modelling, there are several key differences that enable the combination of first class mutable state and higher-order functions.

In Paper I, models are defined for library functions, explaining how to *unlabel* the parameters and *relabel* the result, where the removed labels are stored in a *model state*, which is a map from identifiers to labels. Updates to the models occur when data is being marshaled from the program to the library. With the addition of structured data and higher-order functions, the model state is tied to the wrapper functions via copying when the relabeling operation occurs. Unfortunately, this is not extendable to references, as they require a *shared mutable state*. Consider the following code snippet, where the code above `%%` is the program code and the code below is the library code.

```
1  let (g, r) = lib f 10
2  in r := upg 15 H;
3     g 10
4  %%
5  f x =
6    let r = ref x
7    in (\y . !r, r)
```

The library function creates a reference to `x` on line 6, and uses this for a returned tuple with the first element being a callback function which dereferences the reference, and the second element is the actual reference on line 7. What would be the effect in Paper I when the program code is being executed? On line 1, the program binds the returned tuple from calling `f` to `(g, r)`. It then writes the high value 15 to `r` (line 2), before calling the returned function `g` (line 3). Both values in the tuple should work *on the same reference*. However, Paper I would fail to model this due to the state not being shared. When relabeling, the wrappers for `g` and `r` would be given a copy of the model, making the model update from writing to `r` happen locally in the wrapper for `r`. The end result would be `r` being the secret value 15, and the result of calling `g 10` would be the public value 15, as the value is written to the reference, but the model of `g` is not updated.

The conclusion is that reference models must be shared between all values that have access to the reference. To do this, Paper II moves to a stack/heap based structure. The stack contains pointers to model frames, which reside on the heap. The frames on the heap represent scopes, and form scope chains in combination with the stack. In Paper II, the wrapper functions receives pointers to model frames, which allows for a shared view of the model frames residing on the heap. References and callbacks returned from the library are now tied to the local scope of the function. The example above would now work differently. As the wrappers for `g` and `r` are defined in the same function, the system in Paper II would copy the same frame pointer stack to both wrappers. When the writing to the returned reference occurs on line 2, the written labeled value would be unlabeled, updating the model frame pointed to by the frame pointer with the secret label. As `g` and `r` have copies of the same pointers, the updated model of the reference in the library is seen when `g 10` is called, returning a secret value.

In Paper II, the model state is divided into two parts: the *library model state* and the *call model state*. The library model state contains the information needed to lift a library function to the labeled world, i.e. the library model state is used to lift the library function `f` on line 1. As the lifted library function expects unlabeled parameters when it is invoked, the call model state will hold the labels of the parameters, i.e. the call model state will hold the labels from lines 2 and 3. Everything defined in the library will share the same library model state and the call model states are linked via the stack of pointers. Since the created wrappers can have copies of the same stack of model frame pointers, modifications to the model frames pointed to in the shared library model state residing on the heap is seen by everyone who has a model frame pointer to that model frame. This ensures the same view of the library state, even in the presence of mutability.

6 Bibliography

[1] Adobe: Adobe Acrobat Force-Installed Vulnerable Chrome Extension. https://bugs.chromium.org/p/project-zero/issues/detail?id=1088. accessed: March 2018.

[2] C. S. Advisory. Cisco WebEx Browser Extension Remote Code Execution Vulnerability. https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20170717-webex. accessed: March 2018.

[3] A. Agarwal. Security update and new features. accessed: March 2018.

[4] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *ESORICS*, 2008.

[5] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, 2009.

[6] T. H. Austin and C. Flanagan. Permissive Dynamic Information Flow Analysis. In *PLAS*, 2010.

[7] C. Baraniuk. Ashley Madison: 'Suicides' over website hack. http://www.bbc.com/news/technology-34044506. accessed: March 2018.

[8] BBC News. Adobe hack: At least 38 million accounts breached. http://www.bbc.com/news/technology-24740873. accessed: March 2018.

[9] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical report, 1973.

[10] L. Bello, D. Hedin, and A. Sabelfeld. Value Sensitivity and Observable Abstract Values for Information Flow Control. In *LPAR*, 2015.

[11] D. Chandra and M. Franz. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *ACSAC*, 2007.

[12] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *PLDI*, 2009.

[13] S. Gibbs. Dropbox hack leads to leaking of 68m user passwords on the internet. accessed: March 2018.

[14] J. A. Goguen and J. Meseguer. Security policies and security models. In *S&P*, 1982.

[15] Google. Chrome web store. https://chrome.google.com/webstore/category/extensions?hl=en-GB&_feature=free. accessed: March 2018.

[16] Google. Content Scripts. https://developer.chrome.com/extensions/content_scripts. accessed: March 2018.

[17] Google. Manifest - Web Accessible Resources. https://developer.chrome.com/extensions/manifest/web_accessible_resources. accessed: March 2016.

[18] D. Hausknecht, J. Magazinius, and A. Sabelfeld. May I? - Content Security Policy Endorsement for Browser Extensions. In *DIMVA*, 2015.

[19] D. Hedin, L. Bello, and A. Sabelfeld. Value-Sensitive Hybrid Information Flow Control for a JavaScript-Like Language. In *CSF*, 2015.

[20] INRIA. The Coq Proof Assistant. https://coq.inria.fr/. accessed: March 2018.

[21] J. Keane. Security researcher dumps 427 million hacked Myspace passwords online. https://www.digitaltrends.com/social-media/myspace-hack-password-dump/. accessed: March 2018.

[22] B. Krebs. Online Cheating Site AshleyMadison Hacked. https://krebsonsecurity.com/2015/07/online-cheating-site-ashleymadison-hacked/. accessed: March 2018.

[23] Mozilla. Most Popular Extensions. https://addons.mozilla.org/en-US/firefox/extensions. accessed: March 2018.

[24] J. Pagliery. Hackers selling 117 million LinkedIn passwords. http://money.cnn.com/2016/05/19/technology/linkedin-hack/index.html. accessed: March 2018.

[25] Panopticlick. https://panopticlick.eff.org/.

[26] A. Peterson. https://www.washingtonpost.com/news/the-switch/wp/2014/12/18/the-sony-pictures-hack-explained/?utm_term=.b648dc649bac. accessed: March 2018.

[27] B. Quinn and C. Arthur. PlayStation Network hackers access data of 77 million users. https://www.theguardian.com/technology/2011/apr/26/playstation-network-hackers-data. accessed: March 2018.

[28] O. Räisänen. Trackers leaking bank account data. http://www.windytan.com/2015/04/trackers-and-bank-accounts.html. accessed: March 2018.

[29] A. Russo and A. Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *CSF*, 2010.

[30] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[31] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, pages 40–58, 1999.

[32] Symantec Official Blog. Shellshock: All you need to know about the Bash Bug vulnerability. `https://www.symantec.com/connect/blogs/shellshock-all-you-need-know-about-bash-bug-vulnerability`. accessed: March 2018.

[33] Synopsys. The Heartbleed Bug. `http://heartbleed.com/`. accessed: March 2018.

[34] D. M. Volpano, C. E. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[35] S. A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.